GPGPU/CUDA/C Workshop 2012

Day-2: Intro to CUDA/C Programming

Presenter(s):
Abu Asaduzzaman
Chok Yip

Wichita State University
July 11, 2012
GPGPU/CUDA/C Workshop 2012

Outline
- Review: Day-1
- Brief history of GPGPU and CUDA
- CUDA Development Toolkit
- CUDA Arch&Prog (by Chok)
- Practice:
  - Hello, WSU!
  - Summing Vectors
  - Fun Programming!

QUESTIONS?
Any time, please.
Review: Day-1

(Workshop) Objectives
- To become a moderate to advanced level CUDA/C programmer
- To prepare pedagogy for future CSE courses
- To develop parallel computing research initiatives

Methodologies
- Discuss, study (book?), and practice
- CUDA Educator from Nvidia

(Workshop) Outcomes
- Understand the needs and benefits of parallel programming
- Write program in C, C thread, OpenMP/C, and Open MPI/C
- Understand NVIDIA GPU/CUDA technology
- Develop programs in CUDA/C for GPGPUs
## Workshop Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>9:30 am to 12:00 noon session</th>
<th>1:30 pm to 4:00 pm session</th>
</tr>
</thead>
<tbody>
<tr>
<td>July/10/2012</td>
<td>• Introduction to the Workshop</td>
<td>• Practice</td>
</tr>
<tr>
<td>Tuesday</td>
<td>• Computing: past, present, and future</td>
<td>• C, C threads</td>
</tr>
<tr>
<td>(Asaduzzaman/WSU)</td>
<td>• GPGPU/CUDA/C and WSU</td>
<td>• Open MP/MPI</td>
</tr>
<tr>
<td></td>
<td>• Parallel Computing (by Nasrin)</td>
<td>• Open MPI (SMP, MPI)</td>
</tr>
<tr>
<td>July/11/2012</td>
<td>• Brief history of GPGPU</td>
<td>• Practice</td>
</tr>
<tr>
<td>Wednesday</td>
<td>• Intro to CUDA/C Programming</td>
<td>• Hello, WSU!</td>
</tr>
<tr>
<td>(Asaduzzaman/WSU)</td>
<td>• CUDA Development Toolkit</td>
<td>• Summing vectors</td>
</tr>
<tr>
<td></td>
<td>• CUDA Arch &amp; Prog (by Chok)</td>
<td>• Fun example!</td>
</tr>
<tr>
<td>July/12/2012</td>
<td>• <strong>Thread Cooperation in CUDA/C</strong></td>
<td>• Practice</td>
</tr>
<tr>
<td>Thursday</td>
<td>• Shared memory and synchronization</td>
<td>• Dot products</td>
</tr>
<tr>
<td>(Asaduzzaman/WSU)</td>
<td>• Texture, Page-Locked Host memory</td>
<td>• Matrix multiplication</td>
</tr>
<tr>
<td></td>
<td>• <strong>Advanced CUDA/C Programming</strong></td>
<td>• CUDA/C on multiple GPGPUs</td>
</tr>
<tr>
<td>July/13/2012</td>
<td>• CUDA Threads</td>
<td>• Virtualization on GPGPU</td>
</tr>
<tr>
<td>Friday</td>
<td>• CUDA Memory</td>
<td>• Cloud Computing, MIMD/VLIW, and GPGPU/CUDA/C</td>
</tr>
<tr>
<td>(Ebersole/NVIDIA)</td>
<td>• Performance Considerations</td>
<td>• Thank you!</td>
</tr>
</tbody>
</table>
Review: Day-1 (3)

- Matrix Multiplication
  - C
  - C threads
  - Open MP/C
  - Open MPI/C (shared memory, distributed processors)
  - Open MPI/C (HiPeCC)

- Notice the computing trends
  - Single-thread to multi-thread
  - Single-processor to multi-processor
Outline

- Review: Day-1
- Brief history of GPGPU and CUDA
- CUDA Development Toolkit
- CUDA Arch&Prog (by Chok)

Practice:
- Hello, WSU!
- Summing vectors
- Fun programming!

Questions?
Any time, please.
History of GPGPU and CUDA

- CPU Performance
  - Clock speeds
  - Core count

- Late 1980s and 1990s
  - Graphically driven O/S (Windows 95)
  - 2-D display accelerators
  - In 1992, Silicon Graphics released OpenGL library
  - OpenGL to write 3-D graphics applications
  - More realistic 3-D PC gaming
  - 3-D graphics is consumer computing
  - NVIDIA, ATI Technologies, etc. released graphics accelerators
  - NVIDIA’s GeForce 3 series, Microsoft and DirectX standard
History of GPGPU and CUDA (2)

- Year 2000 and beyond
  - In 2001, NVIDIA’s GeForce 3 series (standard)
  - Programmable vertex and programmable pixel shading stages
  - Developers control to perform computation on GPUs
  - Input colors and texture coordinates to pixel shader → final color
  - Arithmetic is performed on the input color
  - Developer controls the textures
  - Input (colors) can be any (numerical) data

- What did we actually get?
  - SIMD type machines (Array processors, Vector processors)?
  - GPGPU with high arithmetic throughput
  - Programming model too restrictive
History of GPGPU and CUDA (3)

- GPGPU Programming
  - Programming model too restrictive
  - Handful input colors and texture units
  - How and where to write results to memory
  - How to deal with floating-point operations
  - Severe enough? Do you know OpenGL or DirectX?

- In Nov. 2006, NVIDIA GeForce 8800 GTX
  - CUDA Architecture
  - A unified shader pipeline, a programs controls all ALUs
  - ALUs are IEEE standard (not specifically for graphics)
  - Execution units on GPGPUs have read/write access to memory
  - Still you need OpenGL or DirectX to access the new features
History of GPGPU and CUDA (4)

- GeForce 8800 GTX with CUDA Architecture
  - OpenGL or DirectX to access the new features (initially)
  - CUDA C compiler to facilitate programming GPGPUs
  - Hardware driver to exploit the CUDA Architecture

- Applications of CUDA
  - Medical Imaging
  - Computational Fluid Dynamics
  - Environmental Science
History of GPGPU and CUDA (5)

Computing Architecture with CPU and GPU
Peripheral Component Interconnect (PCI)

- CPU
- Memory
- System bus (motherboard)
- PCI bus
- Other peripherals: wireless card, sound card, ...
- Graphics memory
- Video controller
- Graphics processor (GPU)
- Graphics card
History of GPGPU and CUDA (7)

Processing Flow

- Bottlenecks
  - PCI bus: 8GB/s
  - Memory latency: 400-800 clock cycles
  - Memory BW: 192.4 GB/s for GeForce 580

![Diagram of processing flow on CUDA](image)
History of GPGPU and CUDA (8)

Nvidia Tesla

- Nov. 2006: G80, CUDA
- Current: Tesla
  - Tesla C870
  - 518 GFlops
- Tesla: D870
  - Deskside Supercomputer
  - Two Tesla C870 cards in an external unit
  - 1.036 Tflops
- Tesla GPU Server
  - Four G80 cards
  - 2.072 TFlops
CPU - GPU

Pentium Extreme Edition 840
- 3.2 GHz Dual Core
- 230M Transistors
- 90nm process
- 206 mm^2
- 2 x 1MB Cache
- 25.6 GFlops

 GeForce 7800 GTX
- 430 MHz
- 302M Transistors
- 110nm process
- 326 mm^2
- 313 GFlops (shader)
- 1.3 TFlops (total)

(Flops: Floating-point operations)
History of GPGPU and CUDA (10)

CPU - GPU...

Theoretical GFLOP/s

Theoretical GB/s

- NVIDIA GPU Single Precision
- NVIDIA GPU Double Precision
- Intel CPU Single Precision
- Intel CPU Double Precision

- GeForce GTX 380
- GeForce GTX 280
- GeForce 8800 GTX
- Tesla C2050
- Westmere

- GeForce 7800 GTX
- GeForce 6800 Ultra
- Woodcrest
- Tesla C1060
- Bloomfield

- GeForce 6800 GT
- GeForce 5900
- Prescott
- Woodcrest
- Bloomfield
- Harpertown
- Westmere

Dr. Zaman; WSU-5261
Let’s take a break!

Cut out an 8” × 8” square as shown (Left) and rearrange the pieces to form a 5” × 13” rectangle (Right). What is wrong with this?
Outline

- Review: Day-1
- Brief history of GPGPU and CUDA
- CUDA Development Toolkit
- CUDA Arch&Prog (by Chok)
- Practice:
  - Hello, WSU!
  - Summing vectors
  - Fun programming!

QUESTIONS?
Any time, please.
CUDA Development Toolkit

Development Environment

- A CUDA-enabled GPGPU
  - Telsa C2075 (14 procs, 448 cores) and Quadro NVS 295 (1 proc, 8 cores) (in capplab22 PC)
  - GeForce GTX 480 (6 procs, 480 cores) (in capplabmd51 Laptop)

- An NVIDIA device driver
  - To communicate with CUDA-enabled hardware
  - Recompiles kernel module (Linux)
CUDA Development Toolkit (2)

Development Environment

- A CUDA development toolkit and Library (nvcc)
  - CPU compiler and GPU compiler

- A standard C compiler
  - Linux: Debian 6.0 GNOME, typical GNU C compiler (gcc)
CUDA Development Toolkit (3)

Ubuntu: Setting up (by Chok)

- Toolkit and SDK (accordingly)
  - Debian 6: use Ubuntu 10.04 pkg; Fedora 17: use Redhat6.0 pkg
  - chmod +x *.run (x is version)
  - sudo ./cudatoolkit_4.x.x.run
  - ./gpuchemputingsdk_xxxxx.run
Ubuntu: Setting up (by Chok)…

- Install dev-driver
  - `sudo apt-get install kernel-header` (this downloads some files)
  - Now press `<Ctrl>+<Alt>+F1`, login
  - `sudo stop gdm3`
  - `sudo ./devdriver_4.x_linux_64_xxxx.run`
  - Follow on screen instructions
  - `sudo start gdm3`
  - Use filename accordingly
  - Hint: use `<Tab>` key for autocomplete
Outline

- Review: Day-1
- Brief history of GPGPU and CUDA
- CUDA Development Toolkit
- CUDA Arch&Prog (by Chok)

Practice:
- Hello, WSU!
- Summing vectors
- Fun programming!

Questions?
Any time, please.
GPGPU/CUDA/C Workshop 2012

Outline

- Review: Day-1
- Brief history of GPGPU and CUDA
- CUDA Development Toolkit
- CUDA Arch&Prog (by Chok)

Practice:
- Hello, WSU!
- Summing vectors
- Fun programming!

Questions?
Any time, please.

Dr. Zaman; WSU-5261
Access to WSU CAPPLab GPU Servers

- Login to any Linux box in cs network
  - Login in JB205-JB206 Lab; you are already in cs network
  - From outside: ssh userid@kirk.cs.wichita.edu
  - From outside: ssh –X userid@kirk.cs.wichita.edu

- Login to GPU server
  - From cs network: ssh userid@capplab22.cs.wichita.edu
  - From cs network: ssh –X userid@capplabmd51.cs.wichita.edu

- Copy GPU/CUDA/C related files
  - Modify .profile, etc. (to include PATHs)
  - Check gcc, nvcc, etc.
Hello, WSU! (in C)
#include <stdio.h>
void hello_wsu(void) {
    printf("Hello, WSU!\n");
}

int main (void) {
    hello_wsu();
    return 0;
}

$ gcc hello_wsu.c
$ ./a.out
Practice (3)

Hello, WSU! (in CUDA/C)
#include <stdio.h> /* not needed with book.h */
#include “../common/book.h” /* check the PATH */
__global__ void hello_wsu(void) {
    printf(“Hello, WSU!\n”);
}
int main (void) {
    hello_wsu<<1,4>>();
    cudaThreadExit();
    return 0;
}
$ nvcc2 –arch sm_20 hello_wsu.cu
Add two numbers (in C)

```c
#include <stdio.h>
void add2num(int a, int b, int *fun_c) {
    *fun_c = a + b;
}
int main (void) {
    int c; int *main_c;
    add2num(2, 7, main_c);
    main_c = *fun_c;
    printf("c=%d\n", main_c);
    return 0;
}
```
Add two numbers (in CUDA/C)

```c
#include "../common/book.h" /* check the PATH */
__global__ void add2num(int a, int b, int *gpu_c) {
    *gpu_c = a + b;
}
int main (void) {
    int c; int *dev_c;
    cudaMalloc( ... dev_c ...)
    add2num<<<1,1,>>>(2, 7, dev_c);
    cudaMemcpy( ... dev_c ...)
    printf("c=%d\n", dev_c);
    cudaFree(dev_c);
    return 0;
}
```
Hello, WSU! (in CUDA/C) …cont’d

- **Header Files**
  - "../common/book.h"
  - Used with almost for all codes

- **Function Type Qualifiers**
  - **__global__**
  - Called from the host (CPU) only
  - Executed on the device (GPU) → typically called *kernel*
  - Should have void return type
  - Call to a **__global__** must specify the execution configuration

- **HANDLE_ERROR(…)**
  - Detects if the call has returned an error
CUDA/C → extended C

- **Type Qualifiers**
  - global, device, shared, local

- **Keywords**
  - threadIdx, blockIdx

- **Runtime API**
  - Memory, symbol, execution management

- **Function launch**

```
  __global__ void KernelF() {  // Exec D
    __host__ float HostF() {  // Call H
      __device__ float DeviceF() {  // D
        region[threadIdx] = image[i];
      }
    }
  }

  // Allocate GPU memory
  void *myimage = cudaMalloc(bytes)

  // 100 blocks, 10 threads per block
  convolve<<<100, 10>>> (myimage);
```
CUDA/C → extended C…

- **Built-in Variables**
  - `blockIdx = (blockIdx.x, blockIdx.y, blockIdx.y)`
    - three unsigned integers, `uint3`
  - `threadIdx = (threadIdx.x, threadIdx.y, threadIdx.y)`
    - three unsigned integers, `uint3`

- **Built-in Vector types**
  - `dim3`
    - Integer vector type based on `unit3`
    - used to specify dimensions
Practice (9)

Know your device (i.e., GPGPU)
~/CUDAWorkshop2012/Day_2$ vi device_gpu_info.cu

- Name
- Revision
- Clock rate
- Multiprocessors
- Memory
- Registers
- Threads
- Grids
Host, Device, Kernel, Grid, Block, and Thread

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - More…

Courtesy: NDVIA
CUDA Device Memory Allocation

- `cudaMalloc()`
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- `cudaFree()`
  - Frees object from device **Global Memory**
    - Pointer to freed object
CUDA Device Memory Allocation… (another look)
CUDA Host-Device Data Transfer

- `cudaMemcpy()`
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Asynchronous transfer
Practice (14)

Summing vectors

- CUDA/C
Practice (15)

Fun programming!

- CPU/C
Practice (16)

Fun programming!

- GPGPU/CUDA/C
**Practice (17)**

**Matrix Multiplication**

\[
\begin{pmatrix}
  a_{0,0} & a_{0,1} \\
  a_{1,0} & a_{1,1} \\
  a_{2,0} & a_{2,1} \\
  a_{3,0} & a_{3,1}
\end{pmatrix}
\times
\begin{pmatrix}
  b_{0,0} & b_{0,1} & b_{0,2} \\
  b_{1,0} & b_{1,1} & b_{1,2}
\end{pmatrix}
\]

**Result**

\[
\begin{align*}
  c_{0,0} &= a_{0,0} b_{0,0} + a_{0,1} b_{1,0} \\
  c_{0,1} &= a_{0,0} b_{0,1} + a_{0,1} b_{1,1} \\
  c_{0,2} &= a_{0,0} b_{0,2} + a_{0,1} b_{1,2} \\
  c_{1,0} &= a_{1,0} b_{0,0} + a_{1,1} b_{1,0} \\
  c_{1,1} &= a_{1,0} b_{0,1} + a_{1,1} b_{1,1} \\
  c_{1,2} &= a_{1,0} b_{0,2} + a_{1,1} b_{1,2} \\
  c_{2,0} &= a_{2,0} b_{0,0} + a_{2,1} b_{1,0} \\
  c_{2,1} &= a_{2,0} b_{0,1} + a_{2,1} b_{1,1} \\
  c_{2,2} &= a_{2,0} b_{0,2} + a_{2,1} b_{1,2} \\
  c_{3,0} &= a_{3,0} b_{0,0} + a_{3,1} b_{1,0} \\
  c_{3,1} &= a_{3,0} b_{0,1} + a_{3,1} b_{1,1} \\
  c_{3,2} &= a_{3,0} b_{0,2} + a_{3,1} b_{1,2}
\end{align*}
\]
Practice (18)

Matrix Multiplication: \( C[r, c] = A[r, c] \times B[r, c] \)

- GPGPU/CUDA/C
Conclusions

Following topics are covered is Day-2:
- Brief history of GPGPU
- Intro to CUDA/C Programming
- CUDA Development Toolkits
- CUDA Architecture and Programming
- Programming in CUDA/C for GPGPUs

Topics for Day-3:
- Thread Cooperation in CUDA/C
- Texture, Page-Locked Host memory
Questions?

- Any questions, comments, or suggestions?
Thank you.

Please send your feedback to:
Abu Asaduzzaman (Dr. Zaman)
Tel: +1-316-978-5261
E-mail: Abu.Asaduzzaman@wichita.edu